Variability through the Eyes of the Programmer

Jean Melo, Fabricio Batista Narcizo, Dan Witzner Hansen,
Claus Brabrand, Andrzej Wasowski

IT University of Copenhagen, Denmark
Email: {jeanmelo, fabn, witzner, brabrand, wasowski}@itu.dk

***Abstract*—Preprocessor directives (`#ifdefs`) are often used to implement compile-time variability, despite the critique that they increase complexity, hamper maintainability, and impair code comprehensibility. Previous studies have shown that the time of bug finding increases linearly with variability. However, little is known about the cognitive process of debugging programs with variability. We carry out an experiment to understand *how* developers debug programs with variability. We ask developers to debug programs *with* and *without* variability, while recording their eye movements using an eye tracker.**

The results indicate that debugging time increases for code fragments containing variability. Interestingly, debugging time also seems to increase for code fragments without variability in the proximity of fragments that do contain variability. The presence of variability correlates with increase in the number of gaze transitions between definitions and usages for fields and methods. Variability also appears to prolong the "initial scan" of the entire program that most developers initiate debugging with.

***Keywords*-Variability, Preprocessors, Debugging, Eye Tracking, Highly-Configurable Systems**

I. Introduction

Many modern software systems are highly configurable. They embrace variability to increase adaptability, to extend portability across different hardware, and to lower cost. Highly-configurable systems include both large industrial product lines [1], [2], [3] and open-source systems. In some cases, such as the Linux kernel, thousands of configuration options (*features*) controling the compilation process are used [4].

Although bringing important benefits, variability also comes at a cost. It makes reasoning about programs more difficult [5]. As a consequence configuration-dependent (*variability*) bugs appear [6], [7]. Previous studies [5], [8] have shown that debugging is hard and time consuming in the presence of variability. Specifically, our prior study [5] revealed that the time it takes developers to find a bug increases linearly with the number of features, while the ability to actually find the bug is relatively independent of variability. Also, identifying the exact set of configurations in which a bug manifests itself appears to be difficult already for a low number of features.

These prior studies focus on *quantitative* questions only, analyzing debugging time and correctness. There is little evidence in the literature on *how* developers debug programs with variability. In this paper, we describe an eye-tracking experiment with follow-up interviews to study more precisely *how* developers approach and debug programs with variability. We use simplified versions of real buggy programs taken from two highly-configurable systems: BusyBox and BestLap. While the developers debug program versions *with* and *without* variability, we record their eye movements using an eye-tracking device.

Eye movement data constitute a continuous, non-interruptive process measure that can proxy for ongoing mental processes and human activities. Eye movements are intimately linked to the allocation of attention and can be guided by low-level and high-level factors (e.g., [9]). The most commonly studied aspects of eye movement behavior are saccades and fixations, but several additional specialized eye movements that provide different information also exist. Which features attract attention is subject to continued debate and research [9]. Eye tracking has a wide variety of applications outside code comprehension studies. It has been used for medical diagnosis (e.g., Alzheimer's), communication tool for people with severe disability and measures of workload, fatigue and stress levels [10], [9].

Source code comprehension is a multi-level process that involves visual processing, as well as mental encoding and representing the program's source code [11], [12], [13]. Programming involves a series of tasks but with two processes common to all: reading the code (chunking) and searching through the code (tracing). In practice, programmers rarely chunk every statement in a program but the programmer searches for task specific relevant code [11], [13], [14], [15]. Analyzing what the programmer looks at while programming can be monitored through an eye tracker. Eye tracking is used to monitor the eye movements and estimate where the subject is looking (e.g., on a screen) and is typically based on one or more cameras observing the users' eyes [10].

In our study we find that:

- Variability increases debugging time of code fragments that contain variability.
- Debugging time also seems to increase for code fragments without variability in the proximity of fragments that do contain variability.
- Variability makes the number of gaze transitions (also known as *saccades*) grow between definition-usages for fields and call-returns for methods.
- Most developers initiate debugging with an "initial scan" of the program from the first line down to the last, independent of variability. However, variability seems to prolong this "initial scan" of the program disproportionately.
- Developers appear to debug programs with variabil-

```
1  import java.util.Random;
2
3  public class Http {
4      String subject = null;
5      int totalLength = 600;
6      final int HTTP_UNAUTHORIZED = 401;
7      final int HTTP_NOT_IMPLEMENTED = 501;
8      boolean LARGE_FORMAT = false;
9      String REQUEST_GET = "GET";
10
11
12     public void sendHeaders(int responseNum) {
13         if (LARGE_FORMAT) {
14             int buf = 0;
15             buf = totalLength - responseNum;
16             subject = "response header";
17         }
18         if (subject.isEmpty())
19             subject = "Void response";
20         System.out.println("done");
21     }
22
23     private void handleIncoming(String requestType) {
24
25         boolean http_unauthorized = new Random().nextBoolean();
26         if (http_unauthorized)
27             sendHeaders(HTTP_UNAUTHORIZED);
28
29
30         if (!requestType.equals(REQUEST_GET))
31             sendHeaders(HTTP_NOT_IMPLEMENTED);
32
33     }
34
35     public static void main(String[] args) {
36         Http http = new Http();
37         http.handleIncoming("POST");
38     }
39 }
```

(a) *Without* variability.

```
1  import java.util.Random;
2
3  public class Http {
4      String subject = null;
5      int totalLength = 600;
6      final int HTTP_UNAUTHORIZED = 401;
7      final int HTTP_NOT_IMPLEMENTED = 501;
8      #ifdef CONFIG_FEATURE_HTTPD_CGI
9      String REQUEST_GET = "GET";
10     #endif
11
12     public void sendHeaders(int responseNum) {
13         #ifdef CONFIG_LFS
14         int buf = 0;
15         buf = totalLength - responseNum;
16         subject = "response header";
17         #endif
18         if (subject.isEmpty())
19             subject = "Void response";
20         System.out.println("done");
21     }
22
23     private void handleIncoming(String requestType) {
24         #ifdef CONFIG_FEATURE_HTTPD_BASIC_AUTH
25         boolean http_unauthorized = new Random().nextBoolean();
26         if (http_unauthorized)
27             sendHeaders(HTTP_UNAUTHORIZED);
28         #endif
29         #ifdef CONFIG_FEATURE_HTTPD_CGI
30         if (!requestType.equals(REQUEST_GET))
31             sendHeaders(HTTP_NOT_IMPLEMENTED);
32         #endif
33     }
34
35     public static void main(String[] args) {
36         Http http = new Http();
37         http.handleIncoming("POST");
38     }
39 }
```

(b) *With* variability.

Fig. 1: Program `P` *without* and *with* variability.

ity by considering either one configuration at a time (consecutively) or all configurations at the same time (simultaneously).

We hope that our findings will help designers of debugging and developer support tools, and inspire more research to further investigate the interplay between debugging and variability.

II. Motivating Example

Developers of highly-configurable systems often use the C preprocessor (CPP) to implement compile-time variability using conditional compilation directives (`#ifdefs`) [16], [17]. For this reason, we examine the impact of variability on debugging in the context of CPP.

Configurable software systems are challenging for developers because code fragments may be conditionally *included* or *excluded* depending on whether particular features are *enabled* or *disabled*. This means that developers need to reason about several different configurations (versions of the program), each with different data- and control-flow in order to understand a program with variability. This impacts debugging. In programs with variability, some errors occur conditionally, only in certain *erroneous configurations* (i.e., when certain combinations of features are enabled/disabled).

Previous studies have demonstrated that debugging is overall difficult and time consuming in the presence of variability [5], [8]. In this paper, we use eye tracking to study more precisely *how* developers debug programs with variability. We ran a classic "find the bug" experiment using programs containing exactly one error and then compare how the developers look at a program *with* variability against a version of it *without* variability (as a baseline).

Figure 1 presents a code scenario extracted from BUSYBOX which is an open-source highly-configurable system with about 600 features that provides several essential Unix tools in a single executable file. We have adapted the extracted example from C to JAVA to widen the audience of potential participants for our experiment.

Figure 1a shows the version of this program *without* variability derived from the original version with variability shown in Fig. 1b. The program contains an error in line 18 where evaluation of the expression `subject.isEmpty()` causes a *null-pointer exception* because `subject` has the value `null`. The entry point `main` calls `handleIncoming` in line 37 which, in turn, calls `sendHeaders` in line 27. This method then skips past the statements in lines 14–16 because the variable `LARGE_FORMAT` has the value `false` (line 8). Hence, when we reach line 18, the variable `subject` has never been assigned a proper value aside from its initialization to `null` in line 4.

Figure 1b depicts the original version of the program *with* variability. Notice that the program now contains three so-called *features*: `LFS`, `AUTH`, and `CGI` (names abbreviated). Each of these three features can be designated as either *enabled* or *disabled*. Features are used in conditional compilation directives (`#ifdefs`), which control whether to *include* or *exclude* code before compilation, depending on whether features are *enabled* or *disabled*. For instance, the fragment in lines 14–16 (wrapped in an `#ifdef` and `#endif` in lines 13 and 17) is to be *included* in the code if `LFS` is enabled; and *excluded* if `LFS` is disabled. Since n features yield 2^n distinct configurations, our variability

program with three features then comes in eight (2^3) distinct configurations, each corresponding to a different version of the program.

The *null-pointer exception* from before now only appears in specific configurations: whenever we *disable* the feature `LFS` as well as *enable* either `AUTH` or `CGI`. The exception thus occurs in exactly three (out of eight) configurations. The error no longer occurs if we, for instance, *enable* `LFS`; then `subject` is indeed assigned a non-null value in line 16. Also, if we do not *enable* either `AUTH` or `CGI`, `sendHeaders` is no longer invoked in line 27 or 31. The developer must thus somehow consider *all* configurations when debugging a variability program. Further, for a program with variability it is not enough to simply find an error in some configuration. In order to *fix* a bug, a developer must thus not only identify the error, but also correctly identify the exact set of erroneous configurations (combinations of feature enablings/disablings). If the developer gets the configurations wrong, the bug may only be partially fixed. Clearly, this is a difficult task. Combinatorial problems are notoriously difficult.

For these reasons, a developer has to be highly alert and conscious of the features and `#ifdefs` in the code. Previous work has demonstrated to what extent variability complicates debugging. In this paper, we consider *how* variability impacts debugging.

III. Experiment

We have designed a controlled experiment based on eye tracking to investigate and compare how developers debug programs *with* versus *without* variability. We will now explain our experimental design and setup.

A. Objective

The experiment aims to investigate the effect of variability on debugging. In other words, we want to understand *how* developers debug programs with variability in comparison with ordinary programs. Specifically, we aim to answer the following research question:

> **RQ:** How do developers debug programs with variability?

To respond to this question, we perform several so-called *"find the bug"* experiments [18] with an eye tracker and analyze how developers go about finding the bug. We are particularly interested in the impact of variability on bug finding from the developer's perspective.

B. Treatments

We expose each participant to programs *with* and *without* variability, while controlling for noise factors such as learning effect, developer competence, and program complexity. First, we establish programs *without* variability (i.e., simple programs) as a baseline. Then, we consider programs *with* variability containing three features (eight configurations).

C. Participants

We performed the experiment with N=20 participants: seven undergraduate students, one M.Sc. student, seven Ph.D. students, and five post-docs. All participants had programming experience, especially in Java, and around half of the participants had industrial experience ranging from a few months to several years. All subjects were informed that they were free to stop participating at any time, but no one elected to do so.

D. Programs

For the robustness of our experiment—in order to minimize risks of specific effects from a particular program and bug type—we took *two* programs with different kinds of errors. We based our programs on *real* variability errors from two highly-configurable systems: BusyBox and BestLap, taken from previous research [6], [19]. These are qualitatively different systems in terms of size, architecture, purpose, variability, and complexity. BusyBox is an open-source highly-configurable system with 204 KLOC and about 600 features, that provides several essential Unix tools in a single executable file. BestLap is a commercial highly-configurable race game with about 15 KLOC. The kinds of errors we consider are also different: a *null-pointer dereference* and an *assertion violation*. We simplified the error in each system down to an erroneous program that would fit on a screen without scrolling (about 40 lines) yet involve exactly three features.

Bug description of `P`*:* The program has two methods to handle incoming HTTP requests and to send headers (see Fig. 1). As previously explained, the program provokes a null-pointer exception in certain configurations. In debugging this program, the developer needs to identify that the variable `subject` is dereferenced with value null, in exactly three (out of eight) configurations. The method `sendHeaders` is invoked in line 27 (if `AUTH` is *enabled*) or in line 31 (if `CGI` is *enabled*); the variable `subject` will be null whenever the feature `CONFIG_LFS` is *disabled*.

Bug description of `Q`*:* The program originates from a commercial race game. It has one main method responsible for computing a score, as can be seen in the online appendix.[1] The car racing game calculates lap times and assesses qualification for so-called pole position. According to a user requirement, the game should add a penalty when the car crashes. This means that the score can also be negative. However, the method `setScore()` contains a condition prohibiting negative scores. We encode the requirement using assertions. To identify the bug, the developer should somehow see that the variable `totalScore` is always equal to zero after `setScore()` computation, when passing negative values to the method. This error is revealed through an assertion violation in the code (line 31) whenever the features `ARENA` and `NEGATIVE_SCORE` are both enabled, which occurs in exactly two configurations.

Figure 2 lists several characteristics of our benchmark programs. Figure 2a depicts the basic characteristics of the

[1] http://itu.dk/people/jeam/code-gaze-experiment/

Prg	Origin	Filename	Bug type	LOC	#mth
P	BUSYBOX	http.c	*Null-pointer dereference*	39	3
Q	BESTLAP	GameScreen.java	*Assertion violation*	41	4

(a) Basic characteristics.

Prg	#features	Scattering	Tangling	VCC
P	3	4	8	9
Q	3	9	15	14

(b) Variability characteristics.

Fig. 2: Characteristics of our benchmark programs: P and Q.

	Program 1	Program 2
Developer 1	*without* variability	***with* variability**
Developer 2	***with* variability**	*without* variability

Fig. 3: Latin Square (2×2).

programs: project, filename, bug type, number of lines of code, and number of methods (including the `main`). Figure 2b lists the variability characteristics of the programs *with* `#ifdefs`, such as: number of features, feature scattering, feature tangling, and variational cyclometric complexity (VCC). Feature scattering is the number of `#ifdef` blocks throughout the entire program. We put accumulated numbers for all features involved. For example, `P` contains four `#ifdef` blocks. Feature tangling, in turn, counts the number of switches between regular code and feature code or between different features. VCC consists of the *cyclomatic complexity* metric [20] (and counting two flows for `#ifdef` statements).

The programs are similar in terms of size and the number of features. The programs differ in terms of bug type and their variability characteristics, but we control for this in the experiment design, as described below.

E. Design

The two subject programs give rise to four debugging tasks: one for each program *with* and *without* variability. However, in this setup, we need to deal with two main constraints: First, every developer must get each program *once*, otherwise there would be a *learning effect* on subsequent attempts. Second, every developer must get each treatment (with or without variability) *once*, for a similar reason.

Abiding by these constraints, we use a standard *Latin Square design*, which is a common solution for this kind of experiment [21], [22], [23]. A *Latin square* ensures that no row or column contains the same treatment twice. Figure 3 depicts a 2×2 Latin square applied to our context. The columns are labelled with two programs (Program 1 and Program 2). The rows are labelled with two developers (Developer 1 and Developer 2). The four squares in the center contain the two treatments (*with* and *without* variability). Therefore, each developer debugs two *different* programs, each at a *different* variability degree, according to her row.

Finally, we randomly assign participants, treatments, and programs into the Latin squares. The result is the same number of data points for all debugging tasks, without compromising control over the confounding factors such as developer competence or program complexity. For N=20 participants, each performing two out of four debugging tasks, we get exactly 10 data points for each of the four tasks. Technically, our experiment is a *within-group design* in which all participants are exposed to every treatment.

F. Procedure

Before the actual experiment, we executed a *pilot* study with a few local students to test our experiment design and the eye tracking setup. We do not consider the results of the pilot study in our analysis. Based on the pilot study, we optimized mostly the alignment of the programs on the screen for the eye tracker.

The entire experiment consists of five phases: (1) tutorial, (2) warm-up, (3) questionnaire, (4) debugging, and (5) interview. First, when a subject enters the room, we present a tutorial on variability explaining the concepts of features, configurations, and variability. Second, we demonstrate the nature of the tasks and questions through a small warm-up task. Third, we ask the participant to fill in a self-assessment questionnaire about her programming background and experience with JAVA and `#ifdefs`. Fourth, we run the actual debugging experiment using an eye tracker. We use the randomly generated Latin squares to create a *task description sheet* detailing the order of the tasks for a participant. We also run the personal calibration procedure right before the tasks. Then, the participant performs the "find the bug" debugging task for each treatment, in order. Fifth, once a developer finishes the tasks, we conduct a semi-structured interview to get qualitative feedback on how the participant approached the debugging tasks, especially the program with variability. We ask three questions: (i) How did you go about finding the bug? (ii) What were the difficulties? and (iii) How could you fix the bug?

All task description sheets contain instructions and questions that every participant should answer (see the online appendix for an example). We ensure that each program fits onto a single screen to avoid participants scrolling up and down, which would significantly complicate getting the eye tracking data. In other words, we provided the participants with only a static screen (i.e., no IDEs, no tools, no navigation) and the task description sheets on paper. For each participant, we recorded timestamps, the duration of each debugging task, as well as x and y coordinates (fixations) via the eye tracker.

To avoid unintended effects from different software and hardware environments, we executed all experiments on a 64-bit

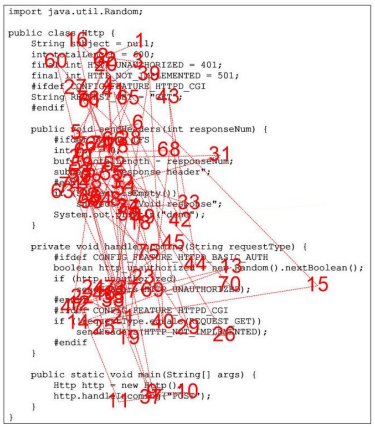


Fig. 4: Fixation sequence: sequence of all fixations from one developer as a "connect-the-dots" visualization.

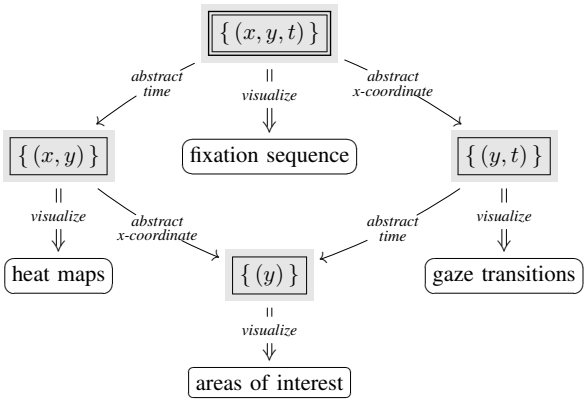


Fig. 5: Overview of abstractions for data analysis.

Windows®8.1 machine, Intel®Core^{TM} i5 CPU running at 2.5 GHz with 8 GB memory. Also, all experiments were conducted in the same room (quiet location). We recorded all of the eye tracking data using the open-source tool OGAMA.[2] In the experiment, we used the Tobii EyeX Controller integrated into the EyeInfo Framework.[3] Together they had a radial accuracy error of < 0.4 degrees which translates to a mere 1.5 lines of inaccuracy on the screen. We discuss implications of this inaccuracy in Sect. V.

We eliminated the data from two subjects because of the eye tracker malfunction occurring during the experiment. The N=20 subjects include only the valid data points. We discuss this issue in Section V. No other deviations happened.

IV. Results & Discussion

We now present the results of the experiment and discuss the implications. We make five observations on how developers debug programs with variability. We begin with presenting our framework of abstractions to simplify data analysis. Then, we address the research question and discuss the findings. All materials of the experiment are available at:

http://itu.dk/people/jeam/code-gaze-experiment/

A. Abstractions for data analysis

Figure 4 displays a scan path registered for a single developer as a sequence of all gaze fixations during a debugging task. For each participant, the eye tracker records a set of triples: x and y coordinates over time t. The raw data enables us to draw the fixation sequence of Fig. 4, a "connect-the-dots" visualization, not particularly helpful for a big set of data of more than one subject. Since the diagram is difficult to understand, we use a range of abstractions (Fig. 5) to ease the data interpretation. We use additional three abstractions besides the fixation sequence, obtained by simplifying away selected dimensions: *heat maps*, *gaze transitions*, and *areas of interest* (AOI).

[2]http://www.ogama.net/
[3]http://eyeinfo.itu.dk/

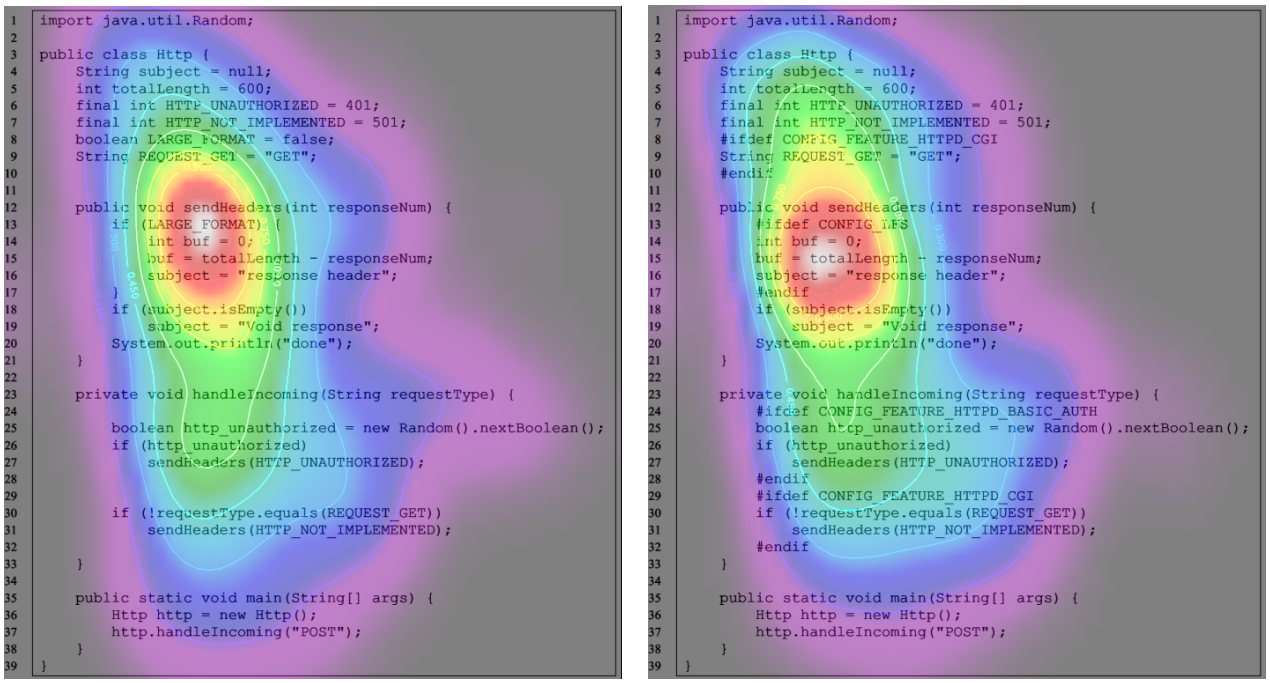
Heat Map: First, we marginalize the data over time which gives us a multi-set of timeless (x, y) pairs. We convert this to a histogram, telling us for each screen location, how much fixation time it attracted. The two-dimensional histogram can be visualized as a so-called *heat map*–using colors to represent the value of time for each location on the screen. An example is shown in Fig. 6. The lowest value in the heat map (lowest fixation time) is shown using the purple color and the highest value is red (long fixation time), with a smooth transition between these extremes.

Gaze Transition (a.k.a. Saccade): We can abstract the x-coordinate away, obtaining a set of (y, t) pairs, from which, we are able to generate a timeline of how the participants read programs vertically—the gaze transitions graphs. In our gaze transition graph (cf. Fig. 11) the y-coordinate is translated to line numbers in code and plotted as a function of time in seconds. The gaze transition graph shows where exactly in the program the participant's gaze is at in the corresponding time.

Areas of Interest: The last abstraction is a combination of the previous two—we abstract away both the horizontal coordinate and time. We end up with a histogram over y-coordinates (see Fig. 8 for an example). This abstraction is primarily useful for displaying the number of fixations related to given *areas of interest* (AOI). An AOI is a region of interest in a study. In our study, we define the areas of interest with the AOI editor in OGAMA following the guidelines provided by Holmqvist et al. [24]. Consequently, once we have the set of y-coordinates and the AOIs defined, we are able to visualize the percentage/amount of fixations from each participant via a table or a bar chart.

An observant reader will notice that we ignored another natural abstraction—marginalizing over the y-coordinate. Abstracting the y-coordinate is not relevant for this study, as we are not interested in how subjects approach lines horizontally (how they read within a line).

We now return to discussion our research question.



(a) *Without* variability. (b) *With* variability.

Fig. 6: Aggregated heat maps for the program P.

	Program P	Program Q
without variability	$5\frac{3}{4}$ min	5 min
with **variability**	$10\frac{1}{2}$ min	$10\frac{1}{2}$ min

Fig. 7: Average total debugging times in our experiment.

B. How do developers debug programs with variability?

Previous work has demonstrated that debugging time increases with variability; in fact, the increase appears to be linear in the number of features [5]. Figure 7 shows the average total debugging time for each of the two programs P and Q, *without* variability (zero features) vs. *with* variability (three features). For both programs, the average total debugging time goes up from roughly *five* to *ten* minutes when the programs involve variability; i.e., the debugging time is *doubled*.

Using the eye tracking data we can investigate deeper *where* developers are spending all this extra debugging time. Based on our eye-tracking experiment, we made five observations:

> OBSERVATION 1: *Variability appears to increase debugging time of the areas of the program that contain variability.*

Figure 6 shows the aggregated heat maps for the program P *without* variability (to the left) versus *with* variability (to the right). Aggregated heat maps are produced by first normalizing (with respect to time) and then superimposing all individual heat maps such that contributions from each developer will be accounted for equally. (Since we have N=20 participants, each aggregated heat map is derived from ten individual heat maps.) Aggregated heat maps give an overall picture of the focus of the developers; i.e., how much they were looking at each part of the program, on average. Importantly, in contrast to Figure 7 that considers *absolute* time, Figure 6 considers *relative* time: how attention is distributed among the program parts.

The *hot spots* (red regions) indicate areas where most of the attention was directed. Not surprisingly, most attention was awarded to the method containing the bug, `sendHeaders` (specifically, lines 12 to 18). Recall that the bug was in line 18 where the condition `subject.isEmpty()` produces a null-pointer exception since the variable `subject` has the value `null`. (In the case *with* variability, this happens in certain configurations.[4]) Overall, the red regions appear quite similar. Without variability, developers dedicate 12% of all fixations to this area (752 out of 6,355). With variability, the dedication to this area is comparable in relative terms with 15% fixations (although using more fixations in absolute terms: 1,249 out of 8,339). The Kullback-Leibler Divergence test confirms that the similarity between the two hot spots is highly significant (divergence value = 0.05, in a scale [0,1]). We observe the same phenomenon for the hot spots in the other program Q (divergence value = 0.07).

Figure 8 details the total time spent looking at each of the four designated *areas of interest* of the program: the field

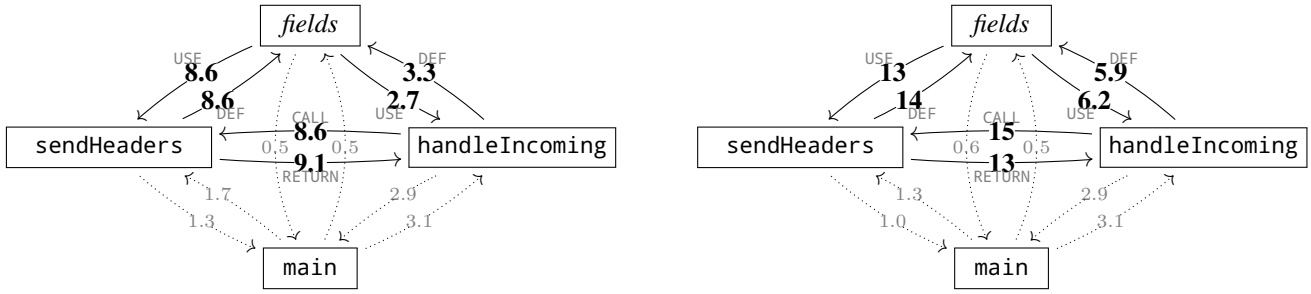
[4]The bug occurs when `LFS` is *disabled* and either `AUTH` or `CGI` is *enabled*; i.e., $\neg$`LFS` $\wedge$ (`AUTH` $\vee$ `CGI`).

area of interest		variability		increase
lines	area	*without*	*with*	factor
4-9	*fields*	26 s	58 s	2.2 x
12-21	`sendHeaders`	63 s	120 s	1.9 x
23-33	`handleIncoming`	56 s	98 s	1.8 x
35-38	`main`	8.2 s	5.3 s	0.7 x
Σ	*all four areas*	153 s	281 s	1.8 x

Fig. 8: Average debugging time for four *areas of interest* of the program `P` *without* vs. *with* variability.

sub-area of interest		variability		increase
lines	sub-area	*without*	*with*	factor
P:12-17	- *with* variability	38 s	77 s	2.0 x
P:18-21	- ***without*** variability	**25 s**	**43 s**	**1.7 x**
Σ	`sendHeaders`	63 s	120 s	1.9 x
Q:18-20	- ***without*** variability	**24 s**	**45 s**	**1.9 x**
Q:21-33	- *with* variability	48 s	130 s	2.7 x
Σ	`gc_computeLevelScore`	72 s	175 s	2.4 x

Fig. 9: Average debugging time for fragments without variability in proximity of fragments with variability.



(a) *Without* variability.

(b) *With* variability.

Fig. 10: Average number of gaze transitions (eye switches) between the differents elements of program `P`.

declarations (lines 4–9); the method `sendHeaders` (lines 12–21); `handleIncoming` (lines 23–33); and `main` (lines 35–38). For instance, the attention devoted to the method `sendHeaders` goes up from about a minute (63 seconds) to two minutes (120 seconds) in the presence of variability; i.e., an increase factor of 1.9 (almost twice as much attention). Overall, it appears that the extra (roughly double) debugging time is spent on all areas of the program that involve variability: the field declarations and the two methods `sendHeaders` and `handleIncoming` all double debugging time. In contrast, no extra time is spent on `main` that does not involve variability. In fact, attention to this area appears to drop slightly in the presence of variability.

Please note that the attention awarded to the four *areas of interest* (last line in Figure 8) does not add up to the total debugging time of Figure 7. This is because the four elements do not cover everything (e.g., imports, blank lines, class definitions, and even areas beyond the screen), gaze transitions (rapid eye movements) are not accounted for in Figure 8, and the total debugging time also involves answering questions about the bugs on a sheet of paper (i.e., not looking at the screen).

> OBSERVATION 2: *Debugging time also increases for code fragments without variability in proximity of code fragments that do contain variability.*

Consider the body of the `sendHeaders` method in program `P` with variability (cf. Fig. 6b). We see that it consists of a code fragment *with* variability (lines 13–17) followed by a fragment *without* variability (lines 18–20). A similar phenomenon occurs in program `Q` in the function `gc_computeLevelScore`, where the top part (lines 18–20) does not contain variability followed by a fragment (lines 21–33) with variability.

Designating these as our *sub-areas of interest*, we can thus zoom in and study the impact of code fragments *with* variability on code fragments *without* variability within the same method.

Figure 9 splits these two methods into their sub-areas of interest with vs. without variability. The sub-areas without variability "in proximity" of variability are shown in bold face. Variability appears to be "contagious" along the flow of control, within a method. Even though lines (18–21) in `P` do not have variability, they go from 25 seconds to 43 seconds to debug in the presence of variability (i.e., debugging takes 1.7 times longer). Similarly, for lines 18–20 in `Q`; they go from 24 seconds to 45 seconds (i.e., debugging takes 1.9 times longer).

We hypothesize that this is because the developers are considering different configurations while debugging (more on this in OBSERVATION 5 later).

> OBSERVATION 3: *Variability appears to increase the number of gaze transitions between definition-usages for fields and call-returns for methods.*

Figure 10 depicts the average number of *gaze transitions* between the four previously introduced areas of interest. Without variability there are, for instance, on average 8.6 navigations from `handleIncoming` to `sendHeaders` and 9.1 back again (see Fig. 10a). Navigations between two methods are annotated with *call* and *return* according to invocations in the program (e.g., `sendHeaders` is called from `handleIncoming` in line 27 and 31). The gaze transition diagrams confirm that the eye movements proceed along method invocations. Similarly,

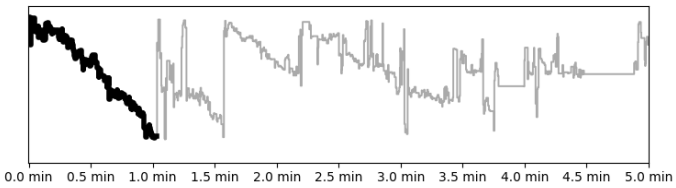


Fig. 11: Gaze transition diagram with *initial scan*.

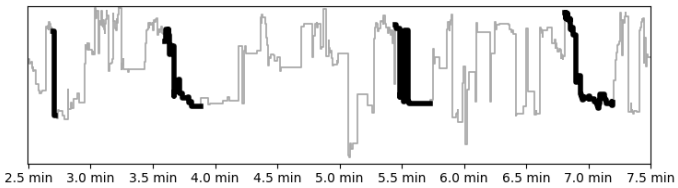


Fig. 12: Gaze transition diagram for a developer using a consecutive strategy and repeatedly considering a method (highlighted in black).

method-to-field navigations are annotated with *def* and *use* as developers navigate from a field variable usage to its *definition* and back again to the *use*. For instance, we see on average 8.6 navigations from `sendHeaders` to the *fields* area of interest (def) and exactly the same number going back again to the usage within the method (use).

With variability, all gaze transitions out of methods containing variability increase significantly (cf. Figure 10b compared to Figure 10a). The *method-to-method* navigation along call-return from `handleIncoming` to `sendHeaders` goes up to 15 and 13 (from 8.6 and 9.1). For *method-to-field*, the (def-use) navigations out of `sendHeaders`, for instance, goes up to 13 and 14 (from 8.6 and 8.6). For navigations out of the method `main` that does not contain any variability (shown as dotted gray edges), we see little change.

Thus, the participants made significantly more gaze transitions in the presence of variability. Again, we hypothesize that developers are exploring and re-exploring different configurations while debugging (cf. OBSERVATION 5).

> OBSERVATION 4: *Variability appears to prolong the "initial scan" of the program (first line to last line) that most developers initiate debugging with.*

Previous work has reported that when performing static code review of program (without variability), reviewers initially perform a preliminary reading of the code, known as a *scan*, whereby a reviewer will *"read the entire code before investigating the details [..]"* [25]. Similarly, when debugging programs (without variability), developers perform a *"first scan followed by several rounds of navigation"* [26].

Not surprisingly, for debugging programs *with* variability, we also see this *initial scan*. Figure 11 illustrates a *gaze transition* diagram of the first five minutes of a participant debugging `P` *with* variability. The diagram shows the y-coordinate the developer is looking at as a function of time. The top of the diagram corresponds to the first line (line 1) and the bottom to the last line of the program (line 39). We have highlighted the initial scan which, in this case, lasted for one minute.

This is supported by the post-experiment interviews: *"I took a global look first from top to bottom and then I started from the main"* (one developer); *"First, I started reading from the top and double checking the fields and methods"* (another one).

Without variability, 7 out of 10 developers performed an initial scan within half a minute (32 seconds) on average. With variability, 8 out of 10 developers scanned initially and it took an average of 51 seconds. Obviously, scanning a larger program will take longer time. Because of the conditional compilation directives (i.e., the `#ifdef` and `#endif` directives), the programs with variability are slightly larger. In fact, they contain 14% more characters. However, this does not account for the 65% increase in debugging time (from 32 to 51 seconds).

> OBSERVATION 5: *Developers appear to debug programs with variability by considering either one configuration at a time (consecutively) or all configurations at the same time (simultaneously).*

The interviews give some qualitative insights into how the subjects debug programs with variability. Most participants complained that they had trouble finding the bug in the presence of variability. One subject explains that he is using a *consecutive strategy* by considering one configuration at a time: *"I began with all features enabled, then I removed one-by-one."* Along the same lines, another explains: *"After I get a good understanding of the code, I started to enable/disable features one at a time to see if the bug appears."* This approach manifests itself on his gaze transition diagram which contains repetitions corresponding to the method `sendHeaders` with variability (cf. Figure 12).

Another subject claims to adopt a *simultaneous strategy* by considering all configurations at the same time: *"I tried to keep track of everything by compiling every combination in mind."* The two strategies are also well-known in *automated* program analysis of programs with variability [27].

Independent of strategies, all developers agreed that debugging programs *without* variability required much less effort. This finding aligns with the study of Medeiros et al. [28] in which they observed that bugs involving variability are easier to introduce and harder to debug and fix than ordinary bugs.

C. Discussion: Implications of our Results

Our results *confirm* previous hypotheses [5], regarding the *accuracy* of debugging programs with variability:

> CONFIRMATION: *Most developers correctly identify bugs in programs with variability; however, many developers fail to identify exactly the set of erroneous configurations (already for 3 features).*

This is also consistent with previous research reporting that developers admit that when fixing programs with variability, they "check only a few configurations of the source code" [28].

Observation 3 (above) stated that developers perform more navigation in the presence of variability. Knowing that, we encourage the programmers using variability to structure the code in a way that minimize the distance between uses and definitions of field variable declarations or between methods calling each other, especially for those declarations and uses that involve `#ifdef`s. At the same time, the builders of development environments shall consider providing convenient ways to navigate from uses to definitions and back again and along call-returns for method invocations. An IDE equipped with continual eye tracking could even automatically "pop up" relevant definitions next to uses as they are being considered by the developer. Clearly, as shown in our data, these pop-ups might be more useful, in areas of code that involve variability (so intensive variability could activate them).

Observations 1–3 indicate that it is worth to contain variability in as few methods as possible to keep other methods variability free. Observation 2 hints that it is advantageous to hoist code fragments without variability "in proximity" of variability out of the method. For instance, in program `P` with variability, lines 18–20 could be moved into a fresh method.

All observations 1–5 may indicate that there are potential gains from *projectional editing* of program with variability. Developers could work separately on particular configurations (programs without variability) which would then be automatically synchonized with the entire variability program (spanning all configurations) [29]. This could be activated/suggested automatically for programmers who work following the consecutive (brute-force) process, as this process can be presumably detected automatically as multiple scans in the eye-tracking data. Of course, we do not know to what extent, or whether at all, these suggestions improve debugging programs with variability. However, our findings do provide indications, that these are the directions that might be worth exploring.

V. Threats to Validity

A. Internal Validity

Selection bias: To minimize selection bias, we randomly assigned our subjects into the Latin squares. Additionally, every participant took all treatments (with and without variability) for all subject programs. Therefore, we controlled the confounding factors via Latin square design and randomization.

All participants voluntarily accepted to take part in the experiment. We found no indications of participants performing deliberately bad or exchanging information. However, we eliminated the data points from two participants because the eye tracker stopped tracking their eye movements during the experiment. For this reason, we did not include them in our data analysis. Thus, the N=20 participants represent only valid data points.

Choice of lab setting: We executed the experiment N=20 times (i.e., one for each participant). All experiments were done in the same room and with our supervision, avoiding extra confounding factors. Since this is the first study of variability debugging using eye-tracking, we opted for a controlled experiment to understand how developers debug programs with variability in lab conditions. We thus optimized for internal validity rather than external validity and a real development environment.

Choice of eye tracker: We used the Tobii EyeX Controller integrated into the EyeInfo Framework for two reasons. First, they are portable and easy to set up and install. Second and, more importantly, they have a good accuracy. In fact, they had a radial accuracy error of < 0.4 degrees only. This is excellent since an accurate and reliable calibration is crucial for eye-tracking studies and conclusions. This translates to an inaccuracy of approximately 1.5 lines on the screen. For this reason, we never considered areas of interest with less than three lines of code.

Choice of language: In this experiment, we used the JAVA programming language because we wanted to run the experiment with several participants and JAVA is well-known among students at our universities. All participants had experience with JAVA, ranging from months to years, including professionally. We chose bugs that are relatively independent of programming language, i.e., they occur in programs written in subset of C that is essentially shared with JAVA.

Choice of the number of features: We studied variability up to *three* features in a program mainly because of timing. Otherwise, the experiment would require much longer time, discouraging and tiring participants. In fact, the participants spent around 15 minutes to debug only the two programs (with and without variability), on average. Additionally, there are very few examples of bugs with higher number of features than three in the literature [6], [30]. Thus, our study focused on the range of variability that seems most relevant.

Program vs. variability complexity: Note that from this experiment and its results, we do not know the *origins* of the extra debugging effort entailed by variability, since we did not focus on the difference between complex programs vs variability programs. This would require another experiment setup and, therefore, it is out of the scope of our study.

B. External Validity

Beyond preprocessors: Our experiment applies to a particular technique for implementing variability: preprocessor (`#ifdefs`). However, among a multitude of technologies that can be used to implement configurable systems, the C preprocessor is one of the oldest, simplest, and most popular mechanisms in use. Generalization to other variability techniques is not intended, even though it might provide hints. Presumably, our results do not translate to CIDE [16], which uses colors to visualize `#ifdef` blocks, since the human visual system is highly sensitive to colors [31].

Beyond university students: The experiment were done predominantly with graduate students. All had JAVA programming experience and several of them had industrial experience. In addition, research has demonstrated that graduate students make good proxies for industry developers [32]. This contributes to representativity and generalization to "real-world" industrial developers. We acknowledge though that more studies

are needed to further understand and confirm the presented observations.

Beyond simple programs: The programs used in our experiment are based on real variability bugs from real highly-configurable systems (BUSYBOX and BESTLAP) and previous research [6], [19], which minimize the risks of studying artificial problems. Additionally, the programs were qualitatively different (cf. Figure 2), and comprising of different kinds of bugs. We thus believe that the results may transfer to other smaller programs.

Beyond simple debugging tasks: We purposely designed our debugging tasks to not require a long time and, consequently, discouraging participants. In fact, the participants spent around 15 minutes to debug the two programs, on average. So, there may be additional effects, unaccounted for, when scaling to longer debugging tasks.

Beyond lab settings: We made sure that each program fits onto a single screen to avoid participants scrolling up and down. We prepared all debugging tasks in slideshow manner using OGAMA, a framework to analyze eye movements. We also did not bold or highlight any code constructs in the programs in order not to attract special attention and, consequently, favor any particular code elements. In other words, no IDEs, no tools, no navigation were provided to the participants; they only had a static screen with the programs and task sheets (on paper). Anything beyond that it is out of the scope for this paper.

Beyond three features: There are very few examples of bugs with higher number of variability than 3 in the literature. Medeiros et al. [30] found that 95% of undeclared/unused bugs involve 0-3 features. Other research also found that variability bugs predominantly involve 0-3 features [6]. For this reason, we focused on this range that seems most relevant. However, it would be interesting to investigate programs with higher number of features.

VI. RELATED WORK

In a previous study, we have investigated the impact of variability on bug finding in terms of time and accuracy [5]. However, our previous study focused only on *quantitative* aspects of debugging, and not on *how* developers debug programs with variability. Thus, to better account for the effect of variability on debugging, we carried out this eye-tracking experiment to actually "see" how developers approach programs with variability, as shown in Section IV. To the best of our knowledge, this is the first study of variability debugging using eye tracking. A few other studies have used eye tracking to study debugging and program comprehension in ordinary programs (i.e., without variability).

Hansen et al. [33] used eye-tracking to investigate factors that impact code comprehension. They found that even subtle notation changes can have impact on performance, and that notation can also make a simple program more difficult to read.

Busjahn et al. [34] conducted eye-tracking studies on small programs to investigate how programmers read code. They found that the fixation durations increased when reading source code in comparison with natural language text. Busjahn et al. [35] also studied linearity and whether or not the linearity effect in reading natural languages transfers to reading of source code. They observed that expert programmers read code less linearly than novices which, in turn, read code less linearly than natural language text.

Rodeghero et al. [36] conducted an eye-tracking study of ten JAVA programmers. They noticed that the programmers looked more at a method's signature than its body in order to summarize it in plain English.

Siegmund et al. [37] conducted a controlled experiment with 17 programmers by applying *functional magnetic resonance imaging* (fMRI) to measure program comprehension. They found a distinct pattern active in five brain regions that are thus deemed necessary for source code comprehension.

None of the these studies investigated debugging in the presence of variability using an eye tracker. In other words, variability was not in their focus. We, in turn, focused on the interplay between debugging and variability from the programmers' perspective. We could draw a number of qualitative conclusions (cf. Section IV). However, we believe that further research using eye tracking on variability debugging is important and required to confront our findings, and to draw new ones.

VII. CONCLUSION

We have presented an experiment aimed at understanding *how* developers debug code with variability implemented using preprocessor directives. We observed that variability increases debugging time for code fragments that contain variability and for neighboring locations. Also, it appears that developers navigate much more between definitions and uses of program objects when interleaved with variability. This is presumably caused by increased complexity of def-use relationships, or by difficulties of maintaining all variants in short-term memory. Variability prolongs the "initial scan" of the program that most subjects initiate debugging with. We notice that developers appear to debug programs with variability by using either a consecutive or simultaneous approach.

Our results are consistent with those of prior studies to the extent that they overlap. The new findings provide some indications how code should be organized to minimize the number of gaze transitions, and on what kind of tools could aid debugging. Automatic tools showing definitions at usage locations, could consider intensive variability as an indicator that the definition is a more sought for information at a given context. Also, possibly, projectional editing techniques can be used to reduce the cognitive overload of variability, especially for subjects using the consecutive approach.

Acknowledgements. We thank Thao for carrying out a pilot study in connection with her Master's Thesis. This work is supported by Brazilian Science without Borders Programme, CNPq grants no. 249020/2013-0 and no. 229760/2013-9. Wasowski is partially funded by The Danish Council for Independent Research, Sapere Aude grant no. 0602-02327B, VARIETE project.

References

[1] P. Clements and L. Northrop, *Software Product Lines: Practices and Patterns*. Addison-Wesley, 2002.

[2] K. Pohl, G. Bockle, and F. J. van der Linden, *Software Product Line Engineering*. Springer, 2005.

[3] T. Berger, D. Nair, R. Rublack, J. M. Atlee, K. Czarnecki, and A. Wasowski, "Three cases of feature-based variability modeling in industry," in *ACM/IEEE 17th International Conference on Model Driven Engineering Languages and Systems (MODELS)*, 2014. [Online]. Available: http://gsd.uwaterloo.ca/sites/default/files/2014-models-vmstudy.pdf

[4] T. Berger, S. She, R. Lotufo, A. Wasowski, and K. Czarnecki, "A study of variability models and languages in the systems software domain," *Software Engineering, IEEE Transactions on*, vol. 39, no. 12, pp. 1611–1640, Dec 2013.

[5] J. Melo, C. Brabrand, and A. Wasowski, "How does the degree of variability affect bug finding?" in *Proceedings of the 38th International Conference on Software Engineering, ICSE 2016, Austin, TX, USA, May 14-22, 2016*, 2016, pp. 679–690. [Online]. Available: http://doi.acm.org/10.1145/2884781.2884831

[6] I. Abal, C. Brabrand, and A. Wasowski, "42 Variability Bugs in the Linux Kernel: A Qualitative Analysis," in *Proceedings of the 29th ACM/IEEE International Conference on Automated Software Engineering*, ser. ASE '14. New York, NY, USA: ACM, 2014, pp. 421–432. [Online]. Available: http://doi.acm.org/10.1145/2642937.2642990

[7] J. Melo, E. Flesborg, C. Brabrand, and A. Wasowski, "A quantitative analysis of variability warnings in linux," in *Proceedings of the Tenth International Workshop on Variability Modelling of Software-intensive Systems*, ser. VaMoS '16. New York, NY, USA: ACM, 2016, pp. 3–8. [Online]. Available: http://doi.acm.org/10.1145/2866614.2866615

[8] S. Schulze, J. Liebig, J. Siegmund, and S. Apel, "Does the discipline of preprocessor annotations matter?: A controlled experiment," in *Proceedings of the 12th International Conference on Generative Programming: Concepts & Experiences*, ser. GPCE '13. New York, NY, USA: ACM, 2013, pp. 65–74. [Online]. Available: http://doi.acm.org/10.1145/2517208.2517215

[9] M. Land and B. Tatler, "Looking and acting: eye movements in everyday life," 2009.

[10] D. W. Hansen and Q. Ji, "In the eye of the beholder: A survey of models for eyes and gaze," *IEEE Transactions on Pattern Analysis and Machine Intelligence*, vol. 32, no. 3, pp. 478–500, March 2010.

[11] M. E. Crosby and J. Stelovsky, "How do we read algorithms? a case study," *Computer*, vol. 23, no. 1, pp. 25–35, 1990.

[12] Y.-G. Guéhéneuc and P. Team, "A theory of program comprehension," 2005.

[13] R. Bednarik and J. Randolph, "Studying cognitive processes in computer program comprehension," in *Passive Eye Monitoring*. Springer, 2008, pp. 373–386.

[14] C. Parnin, "A cognitive neuroscience perspective on memory for programming tasks," *Programming Interest Group*, p. 27, 2010.

[15] M. E. Hansen, A. Lumsdaine, and R. L. Goldstone, "Cognitive architectures: A way forward for the psychology of programming," in *Proceedings of the ACM international symposium on New ideas, new paradigms, and reflections on programming and software*. ACM, 2012, pp. 27–38.

[16] C. Kästner, S. Apel, and M. Kuhlemann, "Granularity in Software Product Lines," in *Proceedings of the 30th International Conference on Software Engineering (ICSE)*. ACM, 2008, pp. 311–320.

[17] J. Liebig, S. Apel, C. Lengauer, C. Kästner, and M. Schulze, "An analysis of the variability in forty preprocessor-based software product lines," in *Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering (ICSE)*. ACM, 2010, pp. 105–114.

[18] P. W. Oman, C. R. Cook, and M. Nanja, "Effects of programming experience in debugging semantic errors," *J. Syst. Softw.*, vol. 9, no. 3, pp. 197–207, Mar. 1989. [Online]. Available: http://dx.doi.org/10.1016/0164-1212(89)90040-X

[19] M. Ribeiro, P. Borba, and C. Kästner, "Feature maintenance with emergent interfaces," in *Proceedings of the 36th International Conference on Software Engineering*, ser. ICSE 2014. New York, NY, USA: ACM, 2014, pp. 989–1000. [Online]. Available: http://doi.acm.org/10.1145/2568225.2568289

[20] T. J. McCabe, "A complexity measure," *IEEE Trans. Softw. Eng.*, vol. 2, no. 4, pp. 308–320, Jul. 1976. [Online]. Available: http://dx.doi.org/10.1109/TSE.1976.233837

[21] R. A. Bailey, *Design of comparative experiment*. Cambridge University Press, 2008.

[22] G. E. P. Box, J. S. Hunter, and W. G. Hunter, *Statistics for Experimenters: design, innovation, and discovery*. Wiley-Interscience, 2005.

[23] D. C. Montgomery, *Design and Analysis of Experiments*. John Wiley & Sons, 2006.

[24] K. Holmqvist, M. Nystrom, R. Andersson, R. Dewhurst, H. Jarodzka, and J. van de Weijer, *Eye Tracking. A comprehensive guide to methods and measures*. Oxford University Press, 2011. [Online]. Available: http://www.oup.com/us/catalog/general/subject/Psychology/CognitivePsychology/CognitivePsychology/?view=usa&ci=9780199697083

[25] H. Uwano, M. Nakamura, A. Monden, and K. ichi Matsumoto, "Analyzing individual performance of source code review using reviewers' eye movement," in *in Proceedings of 2006 symposium on Eye tracking research & applications (ETRA)*, 2006, pp. 133–140.

[26] X. Xie, Z. Liu, S. Song, Z. Chen, J. Xuan, and B. Xu, "Revisit of automatic debugging via human focus-tracking analysis," in *Proceedings of the 38th International Conference on Software Engineering*, ser. ICSE '16. New York, NY, USA: ACM, 2016, pp. 808–819. [Online]. Available: http://doi.acm.org/10.1145/2884781.2884834

[27] C. Brabrand, M. Ribeiro, T. Tolêdo, J. Winther, and P. Borba, "Intraprocedural dataflow analysis for software product lines," *Transactions on Aspect-Oriented Software Development X*, 2013.

[28] F. Medeiros, C. Kästner, M. Ribeiro, S. Nadi, and R. Gheyi, "The love/hate relationship with the C preprocessor: An interview study," in *Proceedings of the 29th European Conference on Object-Oriented Programming (ECOOP)*, ser. Lecture Notes in Computer Science. Berlin/Heidelberg: Springer-Verlag, 2015.

[29] E. Walkingshaw and K. Ostermann, "Projectional editing of variational software," in *Proceedings of the 2014 International Conference on Generative Programming: Concepts and Experiences*, ser. GPCE 2014. New York, NY, USA: ACM, 2014, pp. 29–38. [Online]. Available: http://doi.acm.org/10.1145/2658761.2658766

[30] F. Medeiros, I. Rodrigues, M. Ribeiro, L. Teixeira, and R. Gheyi, "An empirical study on configuration-related issues: Investigating undeclared and unused identifiers," in *Proceedings of the 2015 ACM SIGPLAN International Conference on Generative Programming: Concepts and Experiences*, ser. GPCE 2015. New York, NY, USA: ACM, 2015, pp. 35–44. [Online]. Available: http://doi.acm.org/10.1145/2814204.2814206

[31] D. Moody, "The "physics" of notations: Toward a scientific basis for constructing visual notations in software engineering," *IEEE Trans. Softw. Eng.*, vol. 35, no. 6, pp. 756–779, Nov. 2009. [Online]. Available: http://dx.doi.org/10.1109/TSE.2009.67

[32] R. P. Buse, C. Sadowski, and W. Weimer, "Benefits and barriers of user evaluation in software engineering research," *ACM SIGPLAN Notices*, vol. 46, no. 10, pp. 643–656, October 2011.

[33] M. Hansen, R. L. Goldstone, and A. Lumsdaine, "What makes code hard to understand?" *arXiv preprint arXiv:1304.5257*, 2013.

[34] T. Busjahn, C. Schulte, and A. Busjahn, "Analysis of code reading to gain more insight in program comprehension," in *Proceedings of the 11th Koli Calling International Conference on Computing Education Research*, ser. Koli Calling '11. New York, NY, USA: ACM, 2011, pp. 1–9. [Online]. Available: http://doi.acm.org/10.1145/2094131.2094133

[35] T. Busjahn, R. Bednarik, A. Begel, M. Crosby, J. H. Paterson, C. Schulte, B. Sharif, and S. Tamm, "Eye movements in code reading: Relaxing the linear order," in *2015 IEEE 23rd International Conference on Program Comprehension*, May 2015, pp. 255–265.

[36] P. Rodeghero, C. McMillan, P. W. McBurney, N. Bosch, and S. D'Mello, "Improving automated source code summarization via an eye-tracking study of programmers," in *Proceedings of the 36th International Conference on Software Engineering*. ACM, 2014, pp. 390–401.

[37] J. Siegmund, C. Kästner, S. Apel, C. Parnin, A. Bethmann, T. Leich, G. Saake, and A. Brechmann, "Understanding understanding source code with functional magnetic resonance imaging," in *Proceedings of the 36th International Conference on Software Engineering*, ser. ICSE 2014. New York, NY, USA: ACM, 2014, pp. 378–389. [Online]. Available: http://doi.acm.org/10.1145/2568225.2568252